J Cryptogr Eng (2013) 3:227–240
DOI 10.1007/s13389-013-0061-7

REGULAR PAPER

Strengthening hardware implementations of NTRUEncrypt against fault analysis attacks

Abdel Alim Kamal · Amr M. Youssef

Received: 13 May 2012 / Accepted: 14 May 2013 / Published online: 30 May 2013

Abstract NTRUEncrypt is a parameterized family of lattice-based public key cryptosystems. Similar to other public key systems, it is susceptible to fault analysis attacks. In this paper, we investigate several techniques to strengthen hardware implementations of NTRUEncrypt against this class of attacks. In particular, by utilizing the algebraic structure of the cipher, we propose several countermeasures based on error detection checksum codes, and spatial/temporal redundancies. The error detection capabilities of these countermeasures, as well as their impact on the decryption throughput and area, are also presented.

Keywords NTRUEncrypt · Side channel attacks · Fault analysis countermeasures · Public key cryptography

1 Introduction

The NTRU encryption algorithm, also known as NTRU-Encrypt, is a parameterized family of lattice-based public key cryptosystems [1,2]. Both the encryption and decryption operations of NTRUEncrypt are based on simple polynomial multiplication which makes it very fast compared to other alternatives such as RSA, and elliptic-curve-based systems [3]. Because of its efficiency and promising security, NTRUEncrypt has been accepted to the IEEE P1363 standards under the specifications for lattice-based public-key cryptography. In April 2011, Security Innovation, the company behind the NTRU cryptosystems, announced that its NTRUEncrypt algorithm has been approved by the Accredited Standards Committee X9 as a new encryption standard to protect data for financial transactions.

Fault analysis is an example of side channel attacks in which the attacker is assumed to be able to induce faults in the cryptographic device and observe the faulty output. Then, by careful inspection of the faulty output, the attacker can recover secret information, such as the secret inner state or the secret key. Fault attacks were first introduced by Boneh et al. [4] where they described attacks that target the RSA public key cryptosystem by exploiting a faulty Chinese Remainder Theorem (CRT) computation to factor the public modulus. Subsequently, fault analysis attacks were extended to other digital signature schemes, block ciphers, and stream ciphers (e.g., see [5–7]).

A practical fault analysis attack against NTRUEncrypt was presented in [9]. The used fault model is the one in which the attacker is assumed to be able to fault a small number of coefficients of the polynomial input to (or output from) the second step of the decryption process but cannot control the exact location of injected faults. For the original instantiation of the NTRUEncrypt system with parameters (N, p, q), the attack succeeds with probability $\approx 1 - \frac{1}{p}$ and when the number of faulted coefficients is upper bounded by t, it requires $O((pN)^t)$ polynomial inversions in $\mathbb{Z}/p\mathbb{Z}[x]/(x^N - 1)$.

In this paper, we investigate several techniques to strengthen hardware implementations of NTRUEncrypt against this class of fault analysis attacks. In particular, by utilizing the algebraic structure of the cipher, we propose several countermeasures based on error detection codes, and spatial/temporal redundancy. The impact of these

A. Kamal
Department of Electrical and Computer Engineering (ECE), Concordia University, 1455 De Maisonneuve Blvd. W., Montreal, QC H3G 1M8, Canada
e-mail: a_kamala@encs.concordia.ca

A. M. Youssef (✉)
Concordia Institute for Information Systems Engineering (CIISE), Concordia University, 1455 De Maisonneuve Blvd. W., Montreal, QC H3G 1M8, Canada
e-mail: youssef@ciise.concordia.ca

countermeasures on the throughput and area is investigated and FPGA implementation results are presented.

It should be noted that a fault analysis attack on the NTRUSign digital signature scheme was presented in [8] where the authors also provided some discussion on how to protect NTRUSign from the proposed attack. The main resemblance of NTRUEncrypt and NTRUSign is that both of the two systems rely on the use of the polynomial convolution operations. However, unlike many public key cryptosystems, such as RSA, where the signature scheme is almost identical to the decryption process, this is not the case for the NTRU system where NTRUSign does not bear such a close resemblance to the decryption process in NTRUEncrypt. Furthermore, in [8], the use of error detection codes as a protection mechanism was explicitly excluded since NTRUSign employs a non-linear operation that prevents the possibility of utilizing linear error detection coding schemes. Also, unlike this paper, the work in [8] does not present any hardware implementation and does not provide any concrete analysis in terms of time/area trade-off between the proposed protection methods.

The rest of the paper is organized as follows. In the next section, we briefly review the relevant details of the NTRUEncrypt algorithm. The hardware implementation of the NTRUEncrypt decryption process is described in Sect. 3. The fault analysis attack on NTRUEncrypt is reviewed in Sect. 4 and the proposed countermeasures against this attack are presented in Sect. 5. The hardware implementations of the different countermeasures and their overhead are analyzed in Sect. 6. Section 7 presents some simulation results for the fault coverage of the proposed schemes and provides a some comparison between them. Finally, our conclusion and future works are given in Sect. 8.

2 Description of NTRUEncrypt

The NTRUEncrypt algorithm is a lattice-based public key cryptosystems that is parameterized by three integers: (N, p, q), where N is prime, $\gcd(p, q) = 1$ and $p << q$. Let R, R_p, and R_q be the polynomial rings

$$R = \frac{\mathbb{Z}[x]}{x^N - 1}, \quad R_p = \frac{\mathbb{Z}/p\mathbb{Z}[x]}{x^N - 1}, \quad R_q = \frac{\mathbb{Z}/q\mathbb{Z}[x]}{x^N - 1}.$$

The product of two polynomials $a(x), b(x) \in R$ is given by

$$\begin{aligned} a(x) \star b(x) &= c(x) \\ &= \sum_{k=0}^{N-1} c_k x^k \end{aligned}$$

where

$$c_k = \sum_{i+j \equiv k \bmod N} a_i b_j, \quad 0 \le k \le N-1. \tag{1}$$

For any positive integers d_1 and d_2, let $\tau(d_1, d_2)$ denote the set of ternary polynomials given by

$$\left\{ a(x) \in R : \begin{array}{l} a(x) \text{ has } d_1 \text{ coefficients equal to } 1, \\ a(x) \text{ has } d_2 \text{ coefficients equal to } -1, \\ \text{all other coefficients equal to } 0 \end{array} \right\}$$

In what follows, we briefly describe the key generation, encryption and decryption operations in the NTRU cryptosystem [1].

2.1 Key generation

- Choose a private $f(x) \in \tau(d_f, d_f - 1)$ that is invertible in R_q and R_p.
- Choose a private $g(x) \in \tau(d_g, d_g)$.
- Compute $f_q(x) = f^{-1}(x)$ in R_q and $F_p(x) = f^{-1}(x)$ in R_p.
- Compute $h(x) = f_q(x) \star g(x)$ in R_q.

The polynomial $h(x)$ is the user's public key. The corresponding private key is the pair $(f(x), F_p(x))$. The inverse function of a truncated polynomial can be calculated using the "Almost Inverse Algorithm" presented in [11]. The following steps denote the encryption operations for plaintext $m(x) \in R_p$.

2.2 Encryption

- Choose a random ephemeral key $r(x) \in \tau(d_r, d_r)$.
- Compute the ciphertext $e(x) = pr(x) \star h(x) + m(x) \bmod q$.

2.3 Decryption

- Compute $a(x) = f(x) \star e(x) \bmod q$.
- Compute $b(x)$ =Centerlift$(a(x))$ such that its coefficients lie in the interval $(-q/2, q/2]$.
- Compute $m = F_p(x) \star b(x) \bmod p$.

Table 1 shows some suggested choices for (N, p, q) for different security levels of the original NTRUEncrypt algorithm [10].

Table 1 The parameter sets for NTRU in [10]

	N	p	q	d_f	d_g	d_r
Moderate security	167	3	128	61	20	18
High security	263	3	128	50	24	16
Highest security	503	3	256	216	72	55

3 Hardware implementation of the NTRUEncrypt decryption process

As explained in Sect. 2.3, the decryption operation is performed in three steps: the convolution multiplication $a = f \star e \bmod q$, the Centerlift operation $b = \text{Centerlift}(a(x))$, and finally the convolution multiplication $F_p \star b \bmod p$. Figure 1 shows a hardware architecture that can be used to implement the first step.

In matrix form, the convolution operation $a(x) = e(x) \star f(x) \bmod q$ can be expressed as

$$\begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} e_0 & e_{N-1} & \ldots & e_1 \\ e_1 & e_0 & \ldots & e_{N-2} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ e_{N-1} & e_{N-2} & \ldots & e_0 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \cdot \\ \cdot \\ \cdot \\ f_{N-1} \end{pmatrix}$$

$$= \begin{pmatrix} (f_0 \times e_0 + f_1 \times e_{N-1} + \ldots + f_{N-1} \times e_1) \bmod q \\ (f_0 \times e_1 + f_1 \times e_0 + \ldots + f_{N-1} \times e_{N-2}) \bmod q \\ \cdot \\ \cdot \\ \cdot \\ (f_0 \times e_{N-1} + f_1 \times e_{N-2} + \ldots + f_{N-1} \times e_0) \bmod q \end{pmatrix}. \tag{2}$$

Since $f_j \in \{0, 1, -1\}, 0 \leq j \leq N-1$, then the multiplication $e_i \times f_j$ can be replaced by 0, $+e_i$ and $-e_i$, for $f_j = 0$, 1 and -1, respectively. The value of each coefficient f_j is encoded into two bits, $f_j[0]$ and $f_j[1]$ as follows:

$$\begin{aligned} f_j = 0 &\longrightarrow f_j[1] = 0,\ f_j[0] = 0, \\ f_j = 1 &\longrightarrow f_j[1] = 0,\ f_j[0] = 1, \\ f_j = -1 &\longrightarrow f_j[1] = 1,\ f_j[0] = 1. \end{aligned}$$

Thus, as depicted in Fig. 1, $f_j[1]$ can be used to decide whether the operation performed corresponds to addition or subtraction and $f_j[0]$ can be used to decide whether we

	e_0	e_1	e_2	$\ldots$	e_{N-1}
$\star$					
	f_0	f_1	f_2	$\ldots$	f_{N-1}
+	$e_0 f_0$	$e_1 f_0$	$e_2 f_0$	$\ldots$	$e_{N-1} f_0$
+	$e_{N-1} f_1$	$e_0 f_1$	$e_1 f_1$	$\ldots$	$e_{N-2} f_1$
+	$e_{N-2} f_2$	$e_{N-1} f_2$	$e_0 f_2$	$\ldots$	$e_{N-3} f_2$
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.
+	$e_1 f_{N-1}$	$e_2 f_{N-1}$	$e_3 f_{N-1}$	$\ldots$	$e_0 f_{N-1}$
	a_0	a_1	a_2	$\ldots$	a_{N-1}

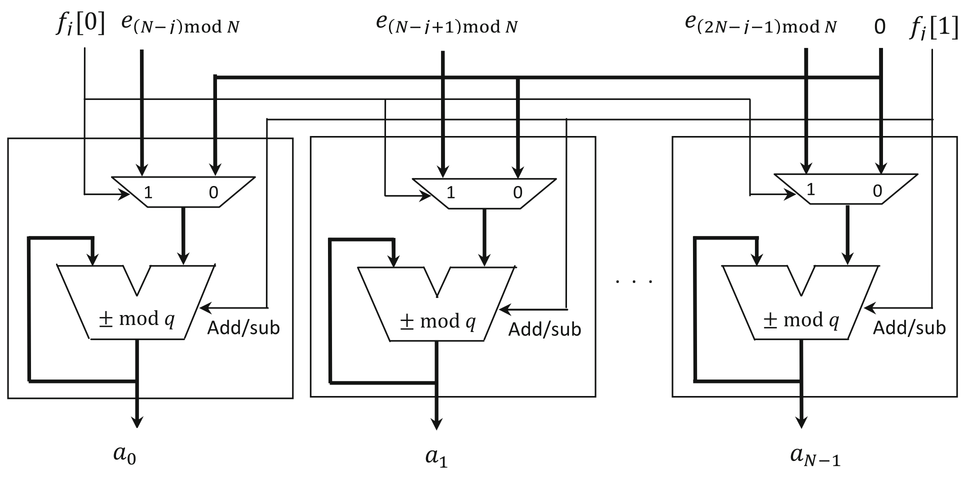
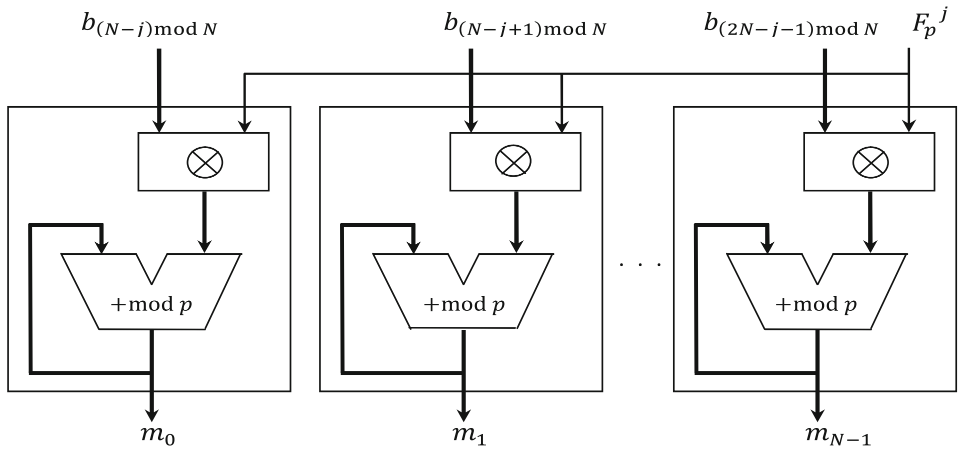


Fig. 1 Hardware architecture for performing the convolution multiplication $f \star e \bmod q$ in N clock cycles

Fig. 2 Hardware architecture for performing the convolution multiplication $F_p \star b$ in N clock cycles

$$\begin{array}{cccccc}
 & b_0 & b_1 & b_2 & \dots & b_{N-1} \\
\star & & & & & \\
 & F_p^0 & F_p^1 & F_p^2 & \dots & F_p^{N-1} \\
\hline
+ & b_0F_p^0 & b_1F_p^0 & b_2F_p^0 & \dots & b_{N-1}F_p^0 \\
+ & b_{N-1}F_p^1 & b_0F_p^1 & bF_p^1 & \dots & b_{N-2}F_p^1 \\
+ & b_{N-2}F_p^2 & b_{N-1}F_p^2 & b_0F_p^2 & \dots & b_{N-3}F_p^2 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots \\
+ & b_1F_p^{N-1} & b_2F_p^{N-1} & b_3F_p^{N-1} & \dots & b_0F_p^{N-1} \\
\hline
 & m_0 & m_1 & m_2 & \dots & m_{N-1}
\end{array}$$



input a 0 to the adder/subtraction module or we input the corresponding e_i coefficient. Since the computation of each coefficient in $a(x)$ requires N clock cycles and since the circuit shown in Fig. 1 performs the computation for all the coefficients at the same time (each rectangle in the bottom part of Fig. 1 performs the computation of one corresponding column in the upper part of the figure), then the final result of the convolution multiplication is produced in N clock cycles. Other hardware architectures that allow the calculation of the convolution operation in less than N clock cycles, at the expense of additional area, were reported in the literature (e.g., see [12–14]). The second step of the decryption process is the Centerlift operation which ensures that the coefficients of $b(x)$ lie in the interval $(-q/2, q/2]$. For step 3, the coefficients of $b(x)$ are reduced mod p and the convolution multiplication $F_p \star b \bmod p$ is calculated as shown in Fig. 2. In this case, the multiplication between the coefficients of $b(x)$ and $F_p(x)$ [also see (6)] has to be performed by a multiplication (modulo p) circuit and cannot be simplified into addition/subtraction as we did for the circuit that calculates $f \star e$.

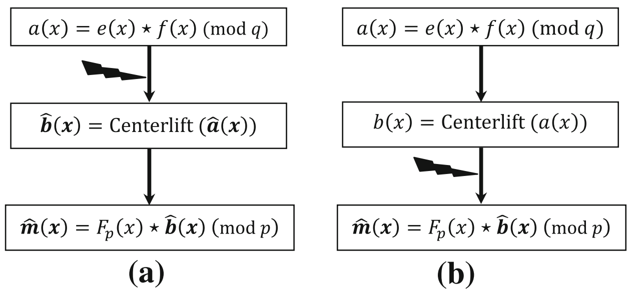
4 Fault analysis of NTRUEncrypt

In this section, we briefly review the attack proposed in [9]. Similar to the case of attacking any public key cryptosystem, this fault attack targets the decryption process which, as explained in the previous section, contains three main operations.

The attacker is assumed to be able to fault a small number of coefficients of the polynomial input to (or output from) the second step of the decryption process but cannot control the exact location of the injected fault. In particular, as depicted in Fig. 3, the attacker is assumed to be able to fault small number of coefficients of the polynomial input to (or output from) the Centerlift operation. Thus, the output of the faulty decryption process can be expressed as:

$$\widehat{m}(x) = F_p \star (b(x) + \epsilon(x)) \bmod p \tag{3}$$

where $\epsilon(x) = \sum_{j=1}^{t} \epsilon_{i_j} x^{i_j} \in R_q, 0 \le i_j < N$, denotes the resulting error polynomial with t non-zero coefficients in the locations corresponding to the injected faults. Thus, the attacker can calculate



$a(x) = e(x) \star f(x) \pmod q$
$\widehat{b}(x) = \text{Centerlift}(\widehat{a}(x))$
$\widehat{m}(x) = F_p(x) \star \widehat{b}(x) \pmod p$
(a)
$a(x) = e(x) \star f(x) \pmod q$
$b(x) = \text{Centerlift}(a(x))$
$\widehat{m}(x) = F_p(x) \star \widehat{b}(x) \pmod p$
(b)

Fig. 3 Modelling the decryption process after inducing faults: **a** before the Centerlift operation or **b** after the Centerlift operation

$$\Delta m(x) = \widehat{m}(x) - m(x) = \epsilon_p(x) \star F_p(x) \bmod p, \tag{4}$$

where $\epsilon_p(x) = \epsilon(x) \bmod p$.

Then, the attacker can obtain candidates for the secret key as

$$f(x) = (\Delta m(x))^{-1} \star \epsilon_p(x) \bmod p \tag{5}$$

by exhaustively trying all possible values for $\epsilon_p(x)$ with the pre-specified small number of non-zero coefficients. The right secret key can be determined by performing the encryption process on the ciphetext and comparing the result with the original plaintext. Further details can be found in [9].

5 Countermeasures

To secure cryptographic devices against fault attacks, proper countermeasures have to be applied to detect any transient or permanent faults and prevent the attacker from accessing the faulty output by immediately disabling the device output or resting all the output bits to 0s. Several techniques of fault detection have been investigated [15]. These techniques include error detection codes and redundancy-based techniques. In what follows, we investigate different approaches to deploy the above fault detection techniques in hardware implementations of NTRUEncrypt and study the trade-off between the fault coverage and the hardware area and throughput overheads.

5.1 Spatial and temporal duplication

Applying spatial duplication to the decryption process is quite straightforward. Spatial duplication requires redundant hardware to allow independent calculations so that faults injected into one hardware unit do not affect (in the same way) the other unit(s).

5.1.1 Decryption–decryption

Figure 4 shows the case where error detection is achieved by duplicating the decryption operation. Throughout our work, we exclude the possibility of higher order faults where the attacker is able to fault the error detection circuity, e.g., by forcing the multiplexer in Fig. 4 to always output m irrespective of the value of its *error* control signal.

While full spatial duplication has practically no impact on throughput and it can detect both permanent and transient faults, the associated area overhead is considerable. Instead, a large saving in the area can be achieved by utilizing temporal redundancy. A naive implementation of this approach would practically double the decryption time. In our implementation, the convolution operation $f \star e$ required by the redundant decryption computation is performed at the same time when the convolution operation $F_p \star b$ is performed since the hardware module that performs the $f \star e$ product is different from the one that performs the $F_p \star b$ product (see the corresponding dotted blue lines in Fig. 8). Thus, the decryption time grows from $\approx 2N$ for the case with decryption-only to $\approx 3N$ for the system with decryption–decryption redundancy.

5.1.2 Decryption–rotation–decryption

Applying temporal redundancy only will not detect permanent faults. The following lemma shows how this limitation can be alleviated by applying the redundant decryption operation on a related ciphertext.

Lemma 1 *Let $e^{(s)}(x) \in R$ denote an s-cyclically shifted version of $e(x) = \sum_{i=0}^{N-1} e_i x^i$, $0 < s \leq N-1$, i.e., $e^{(s)}(x) = (e(x) >>> s)$ (In other words, the coefficients of $e^{(s)}(x)$ are obtained by rotating the coefficients of $e(x)$ by s positions). Then, the plaintext $\grave{m}$ corresponding to the decryption of $e^{(s)}(x)$ is equal to $(m(x) >>> s)$ where $m(x)$ is the plaintext corresponding to the decryption of $e(x)$.*

Proof Since the coefficients of $e^{(s)}(x)$ are obtained by rotating the coefficients of $e(x)$ by s positions, then we have $e^{(s)}(x) = \sum_{i=0}^{N-1} e_i x^{i+s \pmod N} = x^s \star e(x)$.

Let $\grave{a}$ and $\grave{b}$ denote the intermediate computation results during the decryption of $e^{(s)}(x)$. Then, we have

$$\begin{aligned}
\grave{a} &= e^{(s)}(x) \star f(x) \bmod q \\
&= (x^s \star e(x)) \star f(x) \bmod q \\
&= x^s \star (e(x) \star f(x)) \bmod q \\
&= x^s \star a(x) \Rightarrow \\
\grave{b} &= \text{Centerlift}(\grave{a}) \\
&= x^s \star \text{Centerlift}(a(x)) \\
&= x^s \star b(x) \Rightarrow \\
\grave{m} &= \grave{b} \star F_p(x) \bmod p \\
&= (x^s \star b(x)) \star F_p(x) \bmod p \\
&= x^s \star (b(x) \star F_p(x)) \bmod p \\
&= x^s \star m(x).
\end{aligned}$$

□

Thus, in order to detect permanent faults when temporal redundancy is utilized, the redundant computation can be performed using a rotated version of the ciphertext and the two

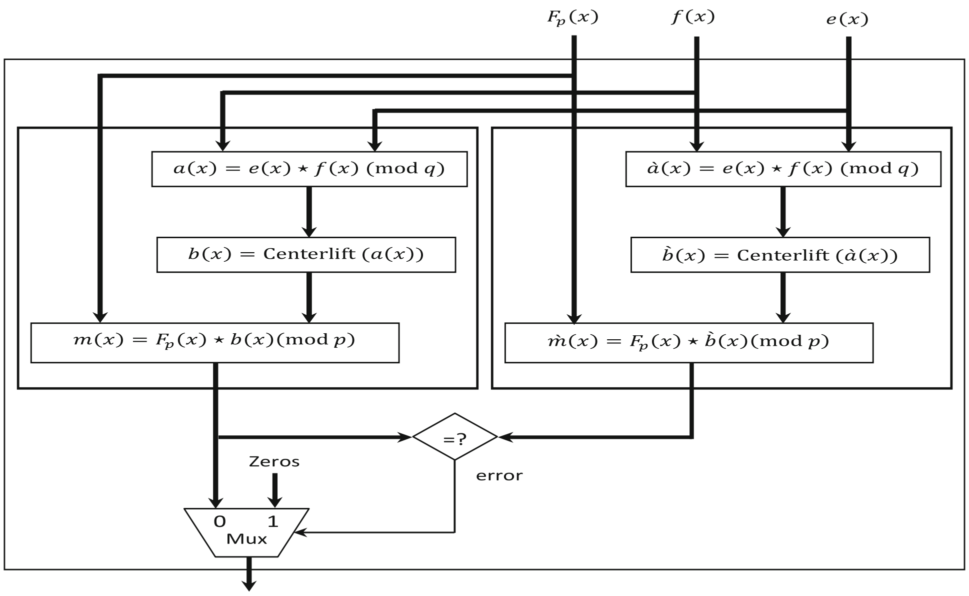

$F_p(x)$
$f(x)$
$e(x)$
$a(x) = e(x) \star f(x) \pmod q$
$b(x) = \text{Centerlift}(a(x))$
$m(x) = F_p(x) \star b(x) \pmod p$
$\grave{a}(x) = e(x) \star f(x) \pmod q$
$\grave{b}(x) = \text{Centerlift}(\grave{a}(x))$
$\grave{m}(x) = F_p(x) \star \grave{b}(x) \pmod p$
=?
Zeros
error
0 1
Mux

Fig. 4 Detecting errors using the duplication method (decryption followed by decryption). When temporal redundancy is used, $\grave{a}(x)$ is calculated at the same time when $m(x)$ is being calculated using the same hardware that is used to calculate $a(x)$

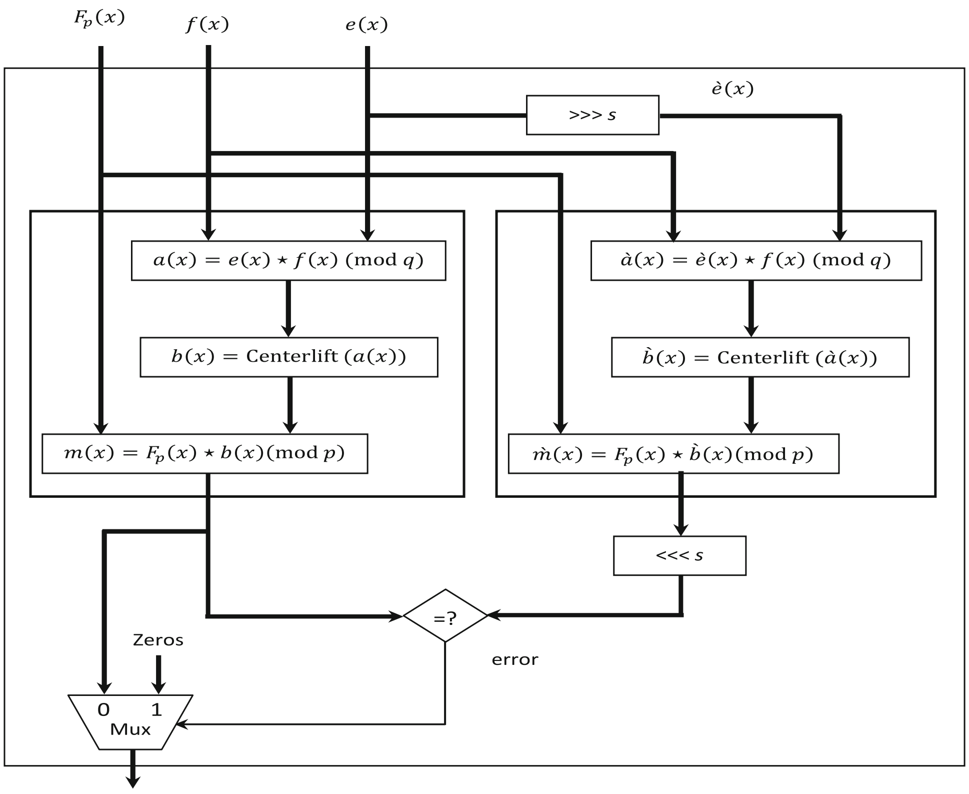

$F_p(x)$
$f(x)$
$e(x)$
$\grave{e}(x)$
>>> s
$a(x) = e(x) \star f(x) \pmod q$
$b(x) = \text{Centerlift}(a(x))$
$m(x) = F_p(x) \star b(x) \pmod p$
$\grave{a}(x) = \grave{e}(x) \star f(x) \pmod q$
$\grave{b}(x) = \text{Centerlift}(\grave{a}(x))$
$\grave{m}(x) = F_p(x) \star \grave{b}(x) \pmod p$
<<< s
=?
Zeros
error
0 1
Mux

Fig. 5 Detecting errors using the duplication method (decryption of a ciphertext and its rotated version)

plaintexts are compared as shown in Fig. 5 where resource sharing and pipelining are used between the two decryption processes in the same way we as did when no rotation was employed.

5.2 Error-detecting codes (EDCs)

The basic idea of code-based fault detection schemes is that a checksum code for the output of a given module can be predicted from the input to this module and consequently it can be compared to the actual checksum code calculated from the output data. The disadvantage of this technique is that corrupted bits may affect the code in such a way that errors may not be detected.

Unlike the case of symmetric key ciphers where traditional parity-based error detection codes [over $GF(2)$] are somewhat straightforward to derive for the intermediate computation steps of the cipher, this does not seem to be the case

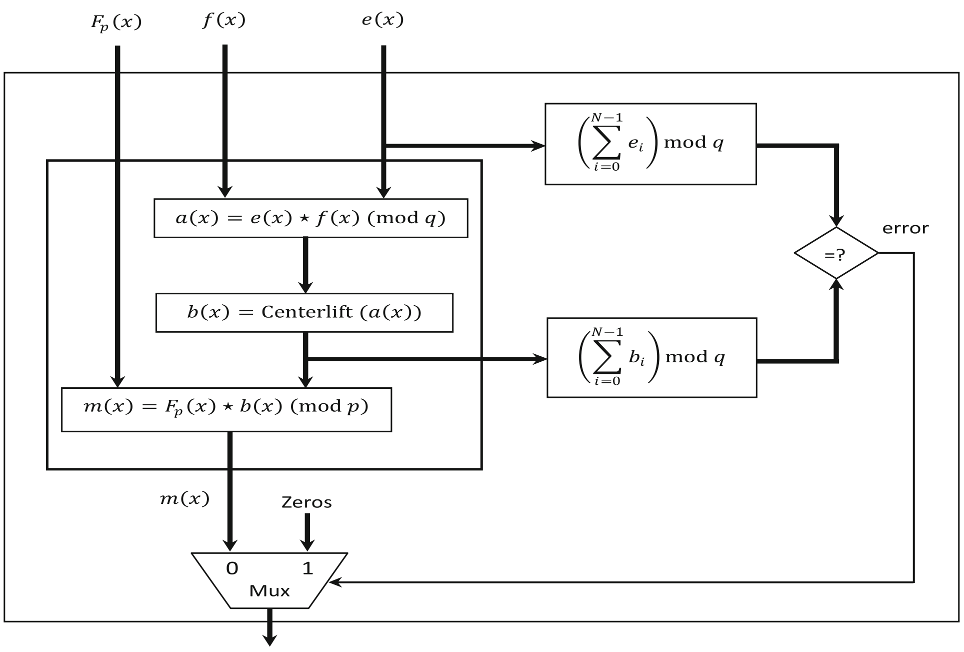

$F_p(x)$
$f(x)$
$e(x)$
$\left(\sum_{i=0}^{N-1} e_i\right) \bmod q$
$a(x) = e(x) \star f(x) \pmod q$
$b(x) = \text{Centerlift}(a(x))$
$\left(\sum_{i=0}^{N-1} b_i\right) \bmod q$
$m(x) = F_p(x) \star b(x) \pmod p$
=?
error
$m(x)$
Zeros
0
1
Mux

Fig. 6 Detecting errors using checksum EDC from Lemma 2

for many public key systems including NTRUEncrypt. On the other hand, the algebraic properties of the polynomials convolutional product allow us to develop simple check sum fault detection techniques which detect both transient and permanent faults.

The following lemmas will be utilized in the development of our EDC based techniques.

Lemma 2 *Let* $e(x) = \sum_{i=0}^{N-1} e_i x^i$, $m(x) = \sum_{i=0}^{N-1} m_i x^i$, *and* $a(x) = \sum_{i=0}^{N-1} a_i x^i, b(x) = \sum_{i=0}^{N-1} b_i x^i$ *denote the ciphertext, plaintext and intermediate computation results as defined in Sect. 2.3. Then, we have*

$$\left(\sum_{i=0}^{N-1} e_i\right) \bmod q = \left(\sum_{i=0}^{N-1} a_i\right) \bmod q = \left(\sum_{i=0}^{N-1} b_i\right) \bmod q$$

Proof In matrix form, the convolution operation $a(x) = e(x) \star f(x) \bmod q$ can be expressed as shown in (2).

Thus, we have

$$\left(\sum_{k=0}^{N-1} a_k\right) \bmod q = \left(\sum_{k=0}^{N-1} e_k \times \sum_{j=0}^{N-1} f_j\right) \bmod q.$$

From the key generation process, we have $f(x) \in \tau(d_f, d_f - 1)$. Thus $\sum_{j=0}^{N-1} f_j = 1$. Consequently, we have

$$\left(\sum_{k=0}^{N-1} a_k\right) \bmod q = \left(\sum_{k=0}^{N-1} e_k\right) \bmod q.$$

The second part of the lemma follows by noting that from the definition of the Centerlift operation, we have $a_i = b_i$ mod q, $i = 0, \ldots, N-1$. □

Figure 6 shows how the checksum equation of Lemma 2 can be applied to implement error detection for errors inserted before or after the calculation of the Centerlift operation (corresponding to Fig. 3a and b, respectively).

Let $\widehat{b}(x) = b(x) + \epsilon(x)$ denote the faulty polynomial corresponding to $b(x)$, where $\epsilon(x) = \sum_{i_j=1}^{t} \epsilon_{i_j} x^{i_j} \in R_q, 0 \le i_j < N$, denotes the resulting error polynomial with t non-zero coefficients in the locations corresponding to the injected faults. From Lemma 2, it is clear that this checksum error detection method cannot detect errors if $\sum_{i_j=1}^{t} \epsilon_{i_j} \equiv 0 \bmod q$. Assuming that the coefficients of the injected errors are uniformity distributed over Z_q and by noting that, in this case, the function $\sum_{i_j=1}^{t} \epsilon_{i_j}$ is a balanced function, i.e., it assumes all possible values in $\{0, 1, \ldots, q-1\}$ with equal probability $= \frac{1}{q}$. Then, by ignoring the possibility of simultaneously faulting the error detection circuit itself, the error detection probability, averaged over all possible values of t, for this scheme is approximately given by $\frac{q-1}{q} = 1 - \frac{1}{q}$.

A mod p checksum formula that holds between the input and output of the third decryption step is given in the Appendix. However, in our implementation, we did not utilize this checksum since it does not improve the overall error detection capability of our implementation. In particular, for the implementation shown in Fig. 6, in order to be able to detect errors that might be injected in $b(x)$ during the calculations of m in the third step of the decryption process, this step is performed on the same set of registers that hold the result of the Centerlift operation mod q. Consequently, any injected errors in the coefficients of $b(x)$ during the calculation of $m(x)$ can be detected, with probability $1 - \frac{1}{q}$, by the checksum in the figure.

In the next section, we show how we can improve the area requirement of this approach without impacting the error detection capability.

5.3 Combining spatial redundancy and error detection codes

While visualizing the the decryption process as composed of three separate steps (see Sect. 2.3) allows us to better understand the mathematics behind it, for hardware implementations, we do not have to separate these three steps. In particular, since the coefficients of b are eventually reduced $\bmod p$ during the calculation of m in the third step of the decryption, one can directly evaluate the Centerlift operation and reduce the coefficient of $b(x) \bmod p$ in one step. In our FPGA prototype, modular reduction $\bmod p$ is implemented using the algorithm for modular reduction mod Mersenne primes (see Algorithm 3.1 in [16]). As depicted in Fig. 7, spatial duplication is applied to detect errors in the mod p operations by duplicating the calculation of Centerlift$(a(x)) \bmod p$ and storing the results in two different registers, b and $\grave{b}$. In this case, by comparing the two registers, any error occurring during the calculation of any of them can be detected with 100 % probability. On the other hand, if the attacker induces a fault in the polynomial $a(x)$, this duplication cannot detect this error because the registers b and $\grave{b}$ will be identical. However, in this case, the applied checksum mod q detects this error with probability $1 - \frac{1}{q}$.

6 FPGA implementation

We used the Xilinx ISE 9.1i framework to design our protected NTRUEncrypt hardware prototype with parameters $(N, p, q, d_f, d_g, d_r) = (167, 3, 128, 61, 20, 18)$. The synthesis was performed with XST application and the simulation was performed using Modelsim. The target FPGA is xcv1000e from Xilinx Virtex-E family.

At the beginning of the decryption operation, the private key $f(x) \in R_p$ is loaded into the chip through N parallel I/O PINS in $T_{\mathrm{ld}(f)} = \lceil \log_2(p) \rceil$ clock cycles. Similarly, the private key $F_p(x)$ is loaded in $T_{\mathrm{ld}(F_p)} = \lceil \log_2(p) \rceil$ clock cycles. The ciphertext $e(x)$ is then loaded in $\log_2(q)$ clock cycles. The most time consuming steps are the convolution operations which require $T_{\mathrm{conv}(\cdot,\cdot)} = N$ clock cycles, each. The Centerlift operation, $T_{\mathrm{cl}(\cdot)}$, consumes one clock cycle. Furthermore, $T_{\mathrm{out}(m)} = \lceil \log_2(p) \rceil$ clock cycles are required to output the plaintext $m(x)$ using N parallel I/O PINS. Let $T_{\mathrm{sum}(a)}$ and $T_{\mathrm{sum}(b)}$ denote the number of clock cycles for accumulating the coefficients of the register $a \bmod q$ and $b \bmod q$, respectively. Figure 8 shows a simplified timeline for the operations required by the above approaches. The figure also illustrates the operations that are performed in parallel and the one that are performed by utilizing resource sharing. In the figure, the time between the vertical dashed lines represents the number of clock cycles required to perform one decryption operation for the input block assuming that the private keys are already loaded into the chip.

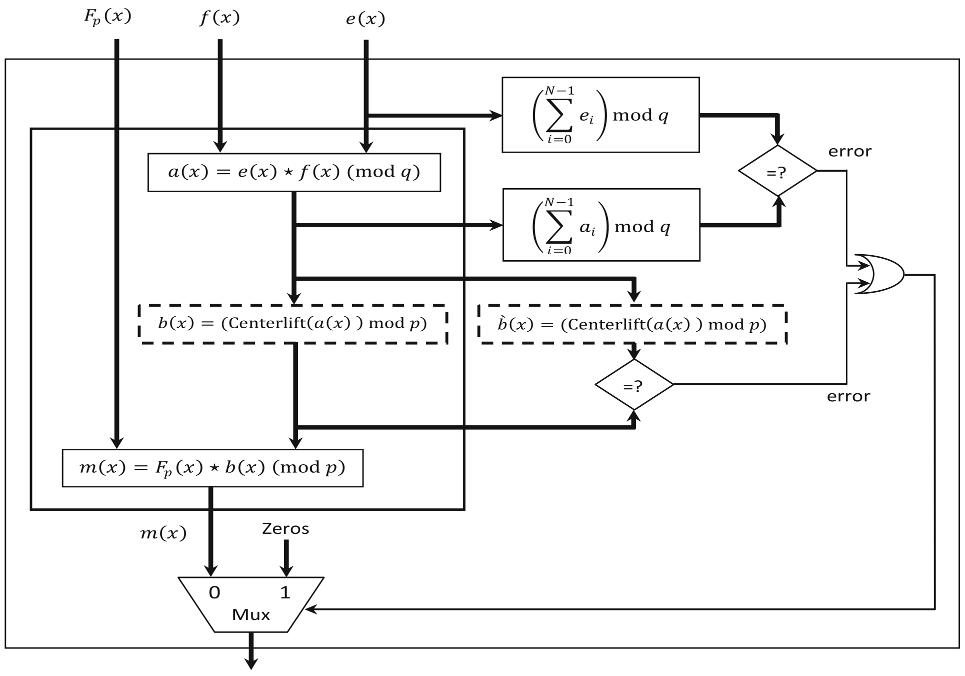

$F_p(x)$
$f(x)$
$e(x)$
$\left(\sum_{i=0}^{N-1} e_i\right) \bmod q$
$a(x) = e(x) \star f(x) \pmod q$
$\left(\sum_{i=0}^{N-1} a_i\right) \bmod q$
=?
error
$b(x) = (\mathrm{Centerlift}(a(x)) \bmod p)$
$\grave{b}(x) = (\mathrm{Centerlift}(a(x)) \bmod p)$
=?
error
$m(x) = F_p(x) \star b(x) \pmod p$
$m(x)$
Zeros
0
1
Mux

Fig. 7 Detecting errors using checksum EDC and spatial redundancy

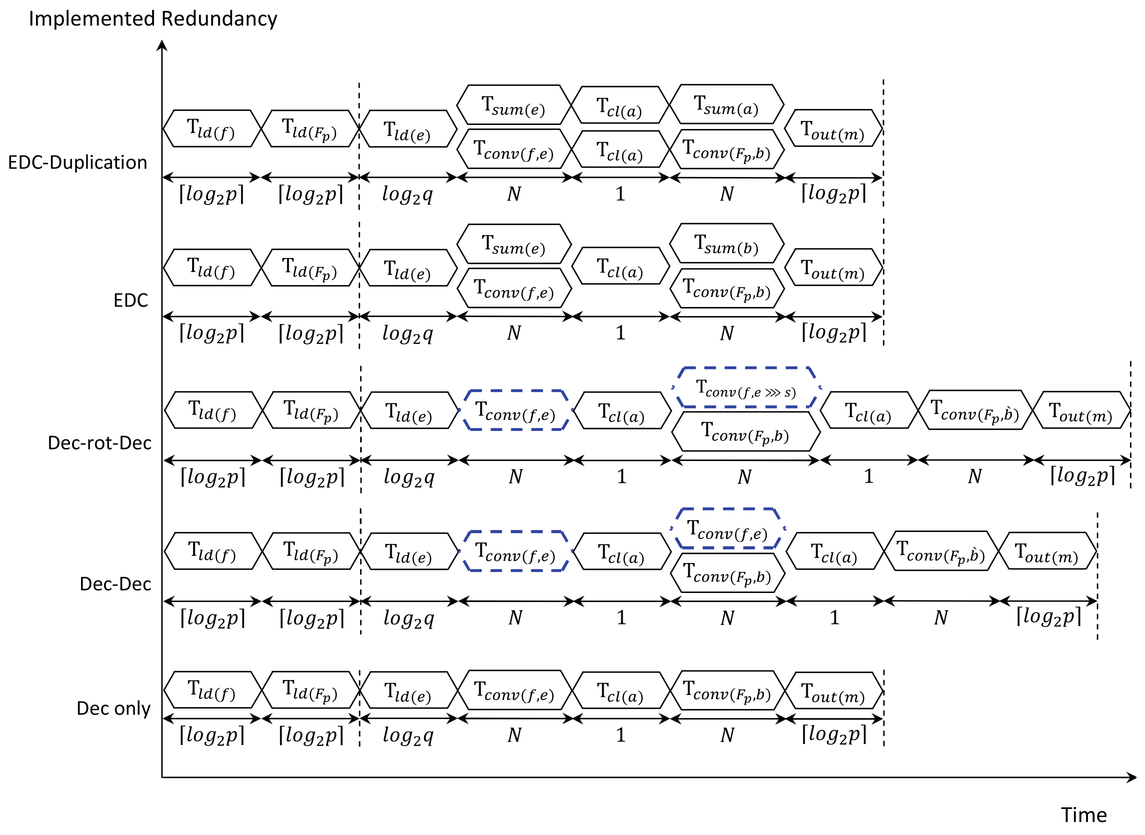


Fig. 8 Time-line (not to *scale*) of operations performed by the proposed architectures. Operations performed using resource sharing are shown in *dotted* (*blue*) *lines* (color figure online)

The IEEE standard 1363.1-2008 specifies a message encoding scheme (from binary to ternary) for NTRUEncrypt. In particular, each three bit quantity of the binary presentation of the message is converted to two ternary coefficients as follows

$$\begin{aligned}&\{0,0,0\}\to\{0,0\},\quad \{0,0,1\}\to\{0,1\},\\&\{0,1,0\}\to\{0,-1\},\ \{0,1,1\}\to\{1,0\},\\&\{1,0,0\}\to\{1,1\},\quad \{1,0,1\}\to\{1,-1\},\\&\{1,1,0\}\to\{-1,0\},\ \{1,1,1\}\to\{-1,1\}.\end{aligned}$$

Given this encoding scheme, each block with N coefficients can be used to encode a message of length $\frac{3}{2}\times N$ bits. Thus

$$\text{Small throughput} \approx \frac{\text{Clock frequency} \times 1.5N}{\text{Clock cycles required to decrypt one block}}.$$

For all designs, the clock frequencies calculated by the synthesis tool were $\gtrsim$55.4 MHz. Table 2 shows the FPGA resources, throughput and error detection capability of all the above-proposed architectures when running at 55 MHz. It should be noted that the above performance figures refer to the raw NTRUEncrypt process. In practice, to encrypt a message securely, one cannot simply convert the message to trinary and apply raw NTRUEncrypt. The message needs to be pre-processed before encryption and post-processed after encryption to protect against active attackers. The necessary processing is specified in IEEE Std 1363.1-2008, and discussed in [17].

7 Comparison of proposed schemes and simulation results for random faults

One of the main objectives of this work is to provide a comparison between different options that can be used to protect NTRUEncrypt against fault analysis attacks. As shown in Table 2, the presented architectures provide different trade off between the time, area and error detection capabilities.

The main advantage of the decryption–decryption method is that it requires a very small area overhead. Under the considered fault model, this method is capable of detecting all transient faults assuming that transient faults are not injected in exactly the same locations during the redundant computation. On the other hand, the decryption–rotation–decryption method is capable of detecting all transient and permanent faults at the expense of increasing the required area. So the two presented methods illustrate different trade-off between

Table 2 FPGA Implementation results for raw NTRUEncrypt decryption with parameters $(N, p, q, d_f, d_g, d_r) = (167, 3, 128, 61, 20, 18)$

	Dec-only	Dec-Dec Fig. 4	Dec-rot-Dec Fig. 5	EDC-only Fig. 6	EDC-duplication Fig. 7
# of slices	3,423	3,636	4,890	4,647	4,260
# of slice FFs	3,919	4,218	4,234	3,579	4,236
# of 4-input LUTs	5,938	6,094	8,457	8,527	7,651
# of IOBs	337	337	337	337	337
#Slices overhead (%)	–	6.22	42.86	35.76	24.45
#Clk cycles per block	$2N + 10$	$3N + 11$	$3N + 11$	$2N + 10$	$2N + 10$
Throughput (K blocks/s)	160	107	107	160	160
Throughput (Mbps)	40.05	26.91	26.91	40.05	40.05
Throughput degradation (%)	–	32.81	32.81	–	–
Type of detected faults	–	Transient	Transient, permanent	Transient, permanent	Transient, permanent
Error detection probability (%)	–	100	100	99.22	99.22

the required area and error detection capabilities. As shown in Table 2, approaches based on error detection codes provide error detection probability ≈99.22 % for the implemented system parameters without acquiring any penalty in the system throughput. While this error detection capability might be suitable for protecting implementations against non-malicious faults, in security sensitive applications where transient faults can be maliciously injected, we believe that this level of protection is not enough; fault attacks are getting better and the number of required faults is getting smaller. Thus, from Table 2, in these situation the system designer can choose between the decryption–decryption approach and the decryption–rotation–decryption approach depending on the design constraints and the assumed attack model, e.g., whether it is required to protect against permanent faults or not.

During the initial phase of this work, we also investigated the use of encryption as a redundancy mechanism to protect against fault attacks on the decryption device. This approach has been applied to the protection of other public key cryptosystems such as RSA [18] where after completing the decryption, the encryption algorithm is applied to the resulting plaintext, and only if the result of the encryption is equal to the original ciphertext, then the system is considered fault-free. For NTRUEncrypt, both $f(x)$ and $F_p(x)$, which represent the private key of the device, can be stored securely on the device. However, this method requires that the decryption device also has access to the public key $h(x)$ as well as to the ephemeral key, $r(x)$, which needs to be freshly generated for each message. Thus, this method cannot be used in practical applications of NTRUEncrypt since $r(x)$ is not available to the decryption device.

In what follows we present the results of our simulations that we performed to validate the error detection capabilities of the proposed methods. The following notation will be used throughout this section.

- N_1 denotes the number of cases where $m(x)$ is not the same as the correct output, i.e., the output in the fault free situation, and the error indication flag cannot detect the erroneous output.
- N_2 denotes the number of cases where $m(x)$ is the same as the correct output but the error indication flag detects an erroneous output.
- N_3 denotes the number of cases where $m(x)$ is not the same as the correct output and the error indication flag detects the erroneous output.
- N_4 denotes the number of cases where $m(x)$ is the same as the correct output and the error indication flag does not detect any error. In other words, such faults are masked and they do not result in any error in the output.

Based on such cases, the fault coverage (FC) is evaluated as a function of the above values as

$$\mathrm{FC} = 100 \times \frac{N_2 + N_3}{N_1 + N_2 + N_3}.$$

Table 3 shows the values for N_1–N_4 and FC when faults occur in one random coefficient in the output of (i) one circuit block (e.g., in the module that calculates $a(x)$) and (ii) in two circuit blocks simultaneously (e.g., in the module that calculates $a(x)$ and the one that calculates $b(x)$, simultaneously). Throughout our simulations, and for random ciphertexts, 1,000 random faults are injected in the selected blocks. As expected, the simulation results for the decryption–decryption case are very similar to the ones obtained for the decryption–rotation–decryption since the fault is assumed to be inserted at a random location in the polynomial presenting the faulted module and consequently the rotation does not affect the results. Also our simulations consider transient faults only, for which both schemes have identical error detection capabilities. When using EDC and the faults are inserted in $m(x)$, the EDC is unable to detect

Table 3 Fault coverage corresponding to faulting one random coefficient of the polynomials in certain blocks (B)

	B	N_1	N_2	N_3	N_4	FC (%)
Dec-Dec	b	0	0	680	320	100.0
	m	0	0	1,000	0	100.0
	$\grave{a}$	0	666	0	334	100.0
	$\grave{b}$	0	670	0	330	100.0
	$\grave{m}$	0	1,000	0	0	100.0
	a, b	0	0	869	131	100.0
	a, m	0	0	1,000	0	100.0
	$a, \grave{a}$	1	199	708	92	99.89
	$a, \grave{b}$	1	212	695	92	99.89
	$a, \grave{m}$	0	328	672	0	100.0
	b, m	0	0	1, 000	0	100.0
	$b, \grave{a}$	2	207	676	115	99.77
	$b, \grave{b}$	0	228	673	99	100.0
	$b, \grave{m}$	0	334	666	0	100.0
	$m, \grave{a}$	0	0	1,000	0	100.0
	$m, \grave{b}$	0	0	1,000	0	100.0
	$m, \grave{m}$	3	0	997	0	99.7
	$\grave{a}, \grave{b}$	0	906	0	94	100.0
	$\grave{a}, \grave{m}$	0	1,000	0	0	100.0
	$\grave{b}, \grave{m}$	0	1,000	0	0	100.0
Dec-rot-Dec	a	0	0	675	325	100.0
	b	0	0	682	318	100.0
	m	0	0	1,000	0	100.0
	$\grave{a}$	0	652	0	348	100.0
	$\grave{b}$	0	680	0	320	100.0
	$\grave{m}$	0	1,000	0	0	100.0
	a, b	0	0	893	107	100.0
	a, m	0	0	1,000	0	100.0
	$a, \grave{a}$	2	226	693	79	99.78
	$a, \grave{b}$	1	197	674	128	99.89
	$a, \grave{m}$	0	313	687	0	100.0
	b, m	0	0	1,000	0	100.0
	$b, \grave{a}$	2	218	684	96	99.78
	$b, \grave{b}$	0	210	687	103	100.0
	$b, \grave{m}$	0	335	665	0	100.0
	$m, \grave{a}$	0	0	1,000	0	100.0
	$m, \grave{b}$	0	0	1,000	0	100.0
	$m, \grave{m}$	0	0	1,000	0	100.0
	$\grave{a}, \grave{b}$	0	886	0	114	100.0
	$\grave{a}, \grave{m}$	0	1,000	0	0	100.0
	$\grave{b}, \grave{m}$	0	1,000	0	0	100.0
EDC-only	a	0	363	637	0	100.0
	b	0	320	680	0	100.0
	m	1,000	0	0	0	0.0
	$\sum e_i$	0	1,000	0	0	100.0

Table 3 continued

	B	N_1	N_2	N_3	N_4	FC (%)
	$\sum b_i$	0	1,000	0	0	100.0
	a, b	8	128	861	3	99.2
	a, m	0	0	1,000	0	100.0
	$a, \sum e_i$	9	303	683	5	99.1
	$a, \sum b_i$	5	323	672	0	99.51
	b, m	0	0	1,000	0	100.0
	$b, \sum e_i$	4	327	666	3	99.6
	$b, \sum b_i$	5	334	658	3	99.5
	$m, \sum e_i$	0	0	1,000	0	100.0
	$m, \sum b_i$	0	0	1,000	0	100.0
	$\sum e_i, \sum b_i$	0	989	0	11	100.0
EDC-duplication	a	0	363	637	0	100.0
	b	0	320	680	0	100.0
	m	1,000	0	0	0	0.0
	$\sum e_i$	0	1,000	0	0	100.0
	$\sum a_i$	0	1,000	0	0	100.0
	$\grave{b}$	0	1,000	0	0	100.0
	a, b	0	131	869	0	100.0
	a, m	0	0	1,000	0	100.0
	$a, \sum e_i$	9	303	683	5	99.1
	$a, \sum a_i$	5	323	672	0	99.5
	$a, \grave{b}$	0	331	669	0	100.0
	b, m	0	0	1, 000	0	100.0
	$b, \sum e_i$	0	341	659	0	100.0
	$b, \sum a_i$	0	350	650	0	100.0
	$b, \grave{b}$	0	313	687	0	100.0
	$m, \sum e_i$	0	0	1,000	0	100.0
	$m, \sum a_i$	0	0	1,000	0	100.0
	$m, \grave{b}$	0	0	1,000	0	100.0
	$\sum e_i, \sum a_i$	0	990	0	10	100.0
	$\sum e_i, \grave{b}$	0	1,000	0	0	100.0
	$\sum a_i, \grave{b}$	0	1,000	0	0	100.0

$N_1 + N_2 + N_3 + N_4 = 1{,}000$ random faults

this error (these corresponds to the rows with FC $= 0.0$). However, it should be noted that this type of faults is not useful to attackers since they better off by just adding random values to the message decrypted with the non-faulted circuit. For the results presented in Table 3, N_2 consists of two sets of faults depending on whether the injected faults are in the original circuit, i.e., the one that computes $m(x)$, or in the overhead part (e.g., in the redundant circuit that computes $\grave{m}(x)$ or in the EDC block). Typically, the faults of the latter set are identified as false alarms. However, this typical definition should be interpreted with care. In particular, system designers should not be tempted to lower this number without carefully considering its meaning. For example, consider the part of the table that corresponds to injecting faults into the decryption–decryption scheme where the faults are injected in the redundant circuit in modules $\grave{a}(x)$, $\grave{b}(x)$ or $\grave{m}(x)$ without affecting the original computation. These faults resulted in corrupting the output of the redundant computation but the original decryption computation still produced the correct output. In this case, N_2 may not be considered as false alarms since the purpose of the error flag is to indicate that the result of the original computation does not agree with the result of the redundant computation even if the result of the original computation agrees with the desired correct decryption. It should also be noted that when protecting cryptographic hardware, it is better to prevent the attacker from accessing the output of the device whenever a fault injection attempt is predicted by the device even if the inserted faults do not change the actual output. Finally, it should be emphasized that the main objective of our work is to provide countermeasures against faults that allow the attacker to recover the secret key of the device and some of the faults considered in the simulations (e.g., faults that affect two blocks simultaneously, faults that only affect the EDC circuitry in Figs. 6, 7 and faults that only affect the block that calculates $m(x) = F_p(x) \star b(x)$ in Figs. 4, 5, 6, 7) do not allow the attacker to do so.

8 Conclusions and future works

We presented different techniques for strengthening the resistance of NTRUEncrypt hardware implementations against fault attacks. We provided a comparison between these different techniques in terms of their error detection capabilities as well as area and throughput overheads. The optimum choice between the proposed methods should be decided by the security engineer depending on the design requirements and the assumed attack model.

While we focused only on FPGA implementations, comparing the space complexities of the presented schemes in terms of gates required for ASICS implementations is a logical extension for this work. Extending fault attacks and developing the corresponding countermeasures for other variants of NTRUEncrypt with $f = 1 + pF_p$, or constructions with padding, are other interesting research directions.

Acknowledgments The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve the quality of the paper. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Appendix

Using the same notation as in Lemma 2, in this appendix, we derive a checksum formula (mod p) that holds between the input and output of the last step in the decryption process.

Lemma 3 *Let $F_p = \sum_{i=0}^{N-1} F_p^i x^i$, where F_p^i is the coefficient of x^i in the polynomial F_p. Then, we have*

$$\left(\sum_{i=0}^{N-1} F_p^i\right) \mathit{mod}\ p = 1.$$

Proof By definition, $f(x) \star F_p$ mod p $= 1$. Thus we have

$$\begin{pmatrix} 1 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{pmatrix} = \begin{pmatrix} f_0 & f_{N-1} & \dots & f_1 \\ f_1 & f_0 & \dots & f_{N-2} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ f_{N-1} & f_{N-2} & \dots & f_0 \end{pmatrix} \begin{pmatrix} F_p^0 \\ F_p^1 \\ \cdot \\ \cdot \\ \cdot \\ F_p^{N-1} \end{pmatrix}$$

$$= \begin{pmatrix} (F_p^0 f_0 + F_p^1 f_{N-1} + \dots + F_p^{N-1} f_1) \bmod p \\ (F_p^0 f_1 + F_p^1 f_0 + \dots + F_p^{N-1} f_{N-2}) \bmod p \\ \cdot \\ \cdot \\ \cdot \\ (F_p^0 f_{N-1} + F_p^1 f_{N-2} + \dots + F_p^{N-1} f_0) \bmod p \end{pmatrix}. \tag{6}$$

Again, by noting that $\sum_{k=0}^{N-1} f_k = 1$, we have

$$\left(\sum_{k=0}^{N-1} F_p^k \times \sum_{j=0}^{N-1} f_j\right) \bmod p = 1 \Rightarrow \left(\sum_{k=0}^{N-1} F_p^k\right) \bmod p = 1.$$

□

Lemma 4

$$\left(\sum_{i=0}^{N-1} b_i\right) \mathit{mod}\ p = \left(\sum_{i=0}^{N-1} m_i\right) \mathit{mod}\ p.$$

Proof Similar to the proof of Lemma 2, we express the convolution operation $m(x) = b(x) \star F_p(x)$ mod p in matrix form. Then, the proof follows by utilizing the result of Lemma 3 to simplify the resulting summation. □

References

1. Hoffstein, J., Pipher, J., Silverman, J.H.: An introduction to mathematical cryptography. Undergraduate texts in mathematics. Springer, New York (2008)
2. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: a ring based public key cryptosystem. In: Buhler, J.P. (ed) Algorithmic number theory symposium-ANTS III, vol. 1423, pp. 267–288. Springer, Heidelberg (1998)
3. Kaliski, B.: Considerations for new public-key algorithms. Netw. Secur. **2000**(9), 9–10 (2000)
4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed) Advances in cryptology-EUROCRYPT'97, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
5. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B. S. Jr. (ed) Advances in cryptology-CRYPTO'97, vol. 1294, pp. 513–525. Springer, Berlin (1997)
6. Hoch, J., Shamir, A.: Fault analysis of stream ciphers. In: Joye, M., Quisquatdr, J.-J. (eds) Cryptographic hardware and embedded systems-CHES'04, vol. 3156, pp. 240–253. Springer, Heidelberg (2004)
7. Biehl, I., Meyer, B., Muller, V.: Differential fault analysis on elliptic curve cryptosystems. In: Bellare, M. (ed) Advances in cryptology-CRYPTO'00, vol. 1880, pp. 131–146. Springer, Berlin (2000)
8. Kamal, A., Youssef, A.M.: Fault analysis of the NTRUSign digital signature scheme. Cryptogr. Commun. **4**(2), 131–144 (2012)
9. Kamal, A., Youssef, A.M.: Fault analysis of the NTRUEncrypt cryptosystem. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E94–A**(4), 1156–1158 (2011)
10. Hoffstein, J., Lieman, D., Pipher, J., Silverman, J.H.: NTRU: a public key cryptosystem. Submissions and contributions to IEEE P1363.1, presented at the August 1999 and October 1999 meetings. http://grouper.ieee.org/groups/1363/lattPK/submissions/ntru.pdf

11. Silverman, J.: Almost inverses and fast NTRU key creation. Report #014, Version 1. http://www.securityinnovation.com/uploads/Crypto/NTRUTech014.pdf. Accessed Mar 2013
12. Atici, A.C., Batina, L., Verbauwhede, I.: Power analysis on NTRU implementation for RFIDs: first results. In: RFID security-RFIDSec'08, pp. 128–139 (2008)
13. Lee, M., Song, J., Choi, D., Han, D.: Countermeasures against power analysis attacks for the NTRU public key cryptosystem. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E93–A**(1), 153–163 (2010)
14. Atici, A.C., Batina, L., Fan, J., Verbauwhede, I., Ors, S.B.: Low-cost implementations of NTRU for pervasive security. In: Application-specific systems, architectures and processors-ASAP, pp. 79–84 (2008)
15. Koren, I., Krishna, C.M.: Fault-tolerant systems. Elsevier/Morgan Kaufmann, Amsterdam (2007)
16. Wilhelm, K.: Aspects of hardware methodologies for the NTRU public key cryptosystem. Thesis submitted to Rochester institute of technology, Rochester, New York (2008)
17. Howgrave-Graham, N., Silverman, J., Singer, A., Whyte, W.: NAEP: provable security in the presence of decryption failures. Cryptology ePrint archive. Report 172 (2003)
18. Joye, M.: Protecting RSA against fault attacks: the embedding method. In: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 41–45 (2009)